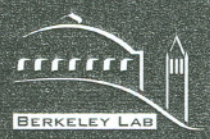

LBNL-40794
UC-405
Preprint

ERNEST ORLANDO LAWRENCE
BERKELEY NATIONAL LABORATORY

An MPI Implementation of the SPAI Preconditioner on the T3E

Stephen T. Barnard, Luis M. Bernardo,
and Horst D. Simon

Computing Sciences Directorate

September 1997

Submitted to
International Journal of Supercomputer Applications

LBNL-40794

An MPI Implementation of the SPAI Preconditioner on the T3E*

Stephen T. Barnard† Luis M. Bernardo‡
Horst D. Simon§

September 8, 1997

Abstract

We describe and test spai_1.1, a parallel MPI implementation of the Sparse Approximate Inverse (SPAI) preconditioner. We show that SPAI can be very effective for solving a set of very large and difficult problems on a Cray T3E. The results clearly show the value of SPAI (and approximate inverse methods in general) as the viable alternative to ILU-type methods when facing very large and difficult problems. We strengthen this conclusion by showing that spai_1.1 also has very good scaling behavior.

1 Introduction

The solution of large, sparse linear systems of equations, obtained from discretization of PDE's, is an important and typical problem in many scientific

*This work was partly supported by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098 and partly (L. Bernardo) supported by the Director, Office of Energy Research, Office of Laboratory Policy and Infrastructure Management, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

†Research Scientist, MJR Technology Solutions, NASA Ames Research Center, Moffet Field, CA 94035 (barnard@nas.nasa.gov)

‡NERSC, MS 50C-3328, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (lmbernardo@lbl.gov)

§NERSC, MS 50B-4230, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (hdsimon@lbl.gov)

and engineering disciplines. Since direct solvers become extremely expensive due to the amount of work and storage required, iterative methods such as CG, GMRES, BICGSTAB, BCG, are typically used [1]. On the other hand, the widespread use of massively parallel computers in scientific applications during recent years has generated, and justified, interest in the development and implementation of efficient parallel algorithms on modern high performance computers. Parallel implementations of these iterative solvers are not difficult to create, but an effective preconditioner is usually required for them to converge in a reasonable number of iterations, or even to converge at all. Unfortunately, the widely used, and effective, ILU-type preconditioners, based on incomplete LU factorizations, are very difficult to parallelize, while the common preconditioners that can be parallelized, such as Polynomial and Block Jacobi, do not seem to be very effective for many important problems. Approximate Inverse preconditioners have been an interesting alternative since they are inherently parallel, and have the potential to be effective too.

The Sparse Approximate Inverse (SPAI) preconditioner, as proposed by Grote and Huckle [2], falls into this category and has already been shown to be effective. The construction of this preconditioner can be expensive compared to ILU-type methods as has been shown in [3, 4] on a number of standard, but rather small, examples. Our results indicate that for very large problems, where ILU-type preconditioners are less efficient, SPAI will become the preconditioner of choice due to its inherent parallelism.

Here we report on spai_1.1, an MPI implementation of SPAI for distributed-memory parallel computers, written by one of the authors (Barnard). The rest of the paper is organized as follows. In Section 2 we review the SPAI algorithm, and in Section 3 we describe spai_1.1 and the techniques used in its implementation (a preliminary version of this work discussed in [5]). Section 4 covers the numerical experiments and Section 5 describes the performance and scaling properties of spai_1.1. In Section 6 a few case studies are discussed and in Section 7 we present our conclusions about SPAI and spai_1.1.

2 SPAI

Consider the system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \mathbf{x}, \mathbf{b} \in \mathbb{R}^n \tag{1}$$

with $\mathbf{A}$ a large, sparse and unsymmetric matrix. We seek a solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. An iterative solver starts with an initial guess $\mathbf{x}_0$ and constructs a sequence $\{\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_m\}$ that is intended to converge to an acceptable approximation $\mathbf{x}_m$ to $\mathbf{x}$ such that $\|\mathbf{r}_m\|/\|\mathbf{b}\| \leq tol$, where $\mathbf{r}_m = \mathbf{b} - \mathbf{A}\mathbf{x}_m$. The convergence is in general not guaranteed, and can be extremely slow. The convergence can however be accelerated by a *preconditioner* $\mathbf{M}$, which can be used either as a right preconditioner,

$$\mathbf{AMy} = \mathbf{b}, \quad \mathbf{x} = \mathbf{My},$$

or left preconditioner,

$$\mathbf{MAx} = \mathbf{Mb}.$$

The matrix $\mathbf{M}$ should be chosen so that $\mathbf{AM}$ (or $\mathbf{MA}$) is a good approximation to the identity $\mathbf{I}$. Here, good approximation is usually understood in the sense of minimizing the Frobenius norm of $(\mathbf{AM} - \mathbf{I})$. This choice naturally leads to inherent parallelism, because the columns $\mathbf{m}_k$ of $\mathbf{M}$ (or the rows in the case of minimizing $\|\mathbf{MA} - \mathbf{I}\|_F$) can be computed independently of one another. In fact, since

$$\|\mathbf{AM} - \mathbf{I}\|_F^2 = \sum_{k=1}^{n} \|(\mathbf{AM} - \mathbf{I})\mathbf{e}_k\|_2^2, \tag{2}$$

the minimization of (2) separates into n independent least squares problems

$$\min_{\mathbf{m}_k} \|\mathbf{A}\mathbf{m}_k - \mathbf{e}_k\|_2, \quad k = 1, ..., n, \tag{3}$$

which can be solved in parallel. Here $\mathbf{e}_k = (0, ..., 0, 1, 0, ..., 0)^T$. The difficulty lies in determining a good sparsity structure for $\mathbf{M}$, so that the solution of (3) yields an effective preconditioner, and a considerable amount of research has already been done in that direction (Yeremin et al. [6, 7, 8], Grote and Simon [9], Cosgrove, Diaz and Griewank [10], Chow and Saad [11], and Grote and Huckle [2]). For the rest of this paper we shall restrict ourselves to SPAI, the method proposed by Grote and Huckle [2], and to spai_1.1, a parallel implementation of SPAI written by one us (Barnard [5]). A closely related version of the parallel SPAI preconditioner is included in ISIS++ [12], which is a an extensive and portable collection of parallel iterative solvers and preconditioners.

2.1 The SPAI Algorithm

Although spai_1.1 constructs a left preconditioner, to be consistent with [2], we briefly describe SPAI as a right preconditioner. The algorithms to construct left or right preconditioners are essentially identical, and one can be converted to the other merely by swapping the meanings of "rows" and "columns" (spai_1.1 constructs a left preconditioner because the matrix-vector multiplication required by iterative methods is most efficiently done on a parallel distributed-memory system when the matrix is distributed row-wise – that is, with complete rows assigned to different processors).

If the sparsity pattern of $\mathbf{M}$ is known then the solution of (3) is straightforward, amounting to the solution of n independent least squares problems. Let $\mathcal{J} = \{j \mid \mathbf{m}_k(j) \neq 0\}$ be the set of indices of the nonzero entries of the kth column of $\mathbf{M}$. The set of indices of rows in $\mathbf{A}$ that could possibly affect a product with column k is $\mathcal{I} = \{i \mid \mathbf{A}(i, \mathcal{J}) \neq 0\}$. To solve (3) we construct the *full* submatrix[1] $\hat{\mathbf{A}} = \mathbf{A}(\mathcal{I}, \mathcal{J})$, which has $|\mathcal{I}|$ rows and $|\mathcal{J}|$ columns, and solve the problem

$$\min_{\hat{\mathbf{m}}_k} \|\hat{\mathbf{A}}\hat{\mathbf{m}}_k - \hat{\mathbf{e}}_k\|_2 \tag{4}$$

where $\hat{\mathbf{e}}_k = \mathbf{e}_k(\mathcal{I})$ and $\hat{\mathbf{m}}_k = \mathbf{M}\mathbf{e}_k(\mathcal{I})$. This can be done, for example, with a QR decomposition as described in [2].

The main difficulty in constructing an approximate sparse inverse is determining the sparsity pattern of $\mathbf{M}$. Grote and Huckle propose the following method. For each column k of $\mathbf{M}$ start with some initial sparsity pattern $\mathcal{J}$, which would typically be diagonal: $\mathcal{J} = \{k\}$. Construct the full submatrix $\hat{\mathbf{A}}$ and solve the least squares problem (4) to obtain $\hat{\mathbf{m}}_k$. Let $\mathbf{m}_k(\mathcal{J}) = \hat{\mathbf{m}}_k$, with the residual

$$\mathbf{r} = \mathbf{A}(., \mathcal{J})\hat{\mathbf{m}}_k - \mathbf{e}_k \ . \tag{5}$$

Assuming that $\|\mathbf{r}\|_2 \neq 0$, then $\mathbf{m}_k$ is not exactly the kth column of the true inverse, and we must *augment* the sparsity structure $\mathcal{J}$ to obtain a better approximation. Therefore look at how to reduce the magnitude of the nonzero components of the residual.

Let $\mathcal{L} = \{l \mid \mathbf{r}(l) \neq 0\}$. Let $\tilde{\mathcal{J}} = \{j \mid \mathbf{A}(\mathcal{L}, j) \neq 0\} \backslash \mathcal{J}$. These are candidate indices to add to $\mathcal{J}$, but there may be very many of them, so it is necessary to somehow choose the ones that most effectively reduce $\|\mathbf{r}\|_2$. Grote and Huckle suggest as a heuristic solving a one-dimensional

[1] Note that we store and operate on $\hat{\mathbf{A}}$ as a dense matrix, although it may contain zero entries.

minimization problem for each $j \in \tilde{\mathcal{J}}$:

$$\min_{\mu_j} \|\mathbf{r} + \mu_j \mathbf{A}\mathbf{e}_j\|_2 \tag{6}$$

which has the solution

$$\mu_j = -\frac{\mathbf{r}^T \mathbf{A}\mathbf{e}_j}{\|\mathbf{A}\mathbf{e}_j\|_2^2} \tag{7}$$

with the residual

$$\rho_j = \|\mathbf{r}\|_2^2 - \frac{(\mathbf{r}^T \mathbf{A}\mathbf{e}_j)^2}{\|\mathbf{A}\mathbf{e}_j\|_2^2} \tag{8}$$

The procedure for choosing new indices to augment the sparsity structure $\mathcal{J}$ is as follows:

1. Determine $\tilde{\mathcal{J}}$,
2. Determine ρ_j for all $j \in \tilde{\mathcal{J}}$,
3. Determine the mean of $\{\rho_j\}$,
4. Retain all indices in $\tilde{\mathcal{J}}$ corresponding to a value of ρ less than or equal to the mean, up to to some maximum number of indices (typically 5).

The algorithm stops when either a maximum number of fill-ins (nonzero entries) per column is reached or the condition

$$\|\mathbf{r}\|_2 < \epsilon \tag{9}$$

is satisfied, where $0 < \epsilon < 1$, is a parameter that determines the accuracy of the sparse-inverse approximation. A more detailed description of the SPAI algorithm is given in [5].

3 spai_1.1, an MPI Implementation of SPAI

Although SPAI is an inherently parallel algorithm, there are several difficult issues to confront in creating an efficient and portable implementation. These issues were the main topic in [5], but for the sake of completeness we describe them here again.

3.1 One-Sided Communication

SPAI computes every row of $\mathbf{M}$ independently, but to do so it must access potentially any row of $\mathbf{A}$ in a completely unpredictable way. A processor that computes a row of $\mathbf{M}$ must therefore access rows of $\mathbf{A}$ that reside on other processors. This is straightforward on a shared-memory architecture, but on a distributed-memory system with no support for shared-memory programming it requires either expensive and nonscalable all-to-all communication or so-called "one-sided" communication. We use MPI for maximum portability, but MPI does not support one-sided communication directly. It does, however, provide the functionality to implement one-sided communication in a specialized way.

The processors computing rows of $\mathbf{M}$ run entirely asynchronously, with no barriers until $\mathbf{M}$ is completed. Whenever a processor needs access to data on another processor, or when it needs to inform another processor of some condition, it sends a request to that processor in the form of a short message. These requests are handled by a *communications server* that uses the `MPI_Iprobe` function to detect the arrival of requests.

There are five types of requests, distinguished by their message tags in the communications server:

1. Another processor needs a row of $\mathbf{A}$.

2. Another processor needs a row of $\mathbf{M}$. This is part of the load balance mechanism described below.

3. Another processor is storing a row of $\mathbf{M}$. Again, this is part of the load balancing mechanism.

4. A processor has finished constructing all the rows of $\mathbf{M}$ that it "owns" and is informing the master processor that it has finished its local work (although it may still construct rows owned by other processors until all processors have finished their local work).

5. The master processor informs all other processors that the construction of $\mathbf{M}$ has been completed.

The communications server is called periodically by every processor, typically when they are waiting for remote data or when they have finished a substantial amount of work, such as computing a row of $\mathbf{M}$.

3.2 Latency Hiding

Many distributed-memory computers have large latency in interprocessor communication. The parallel spai_1.1 code masks this latency as much as possible by using asynchronous communication and overlapping work with communication. For example, when a processor initiates a request for a row of $\mathbf{A}$ to another processor it uses the asynchronous `MPI_Isend` function, then it repeatedly calls the communications server to service requests from other processors until the data that it requested arrives.

One effective way that the parallel spai_1.1 code hides latency is to avoid unnecessary communication altogether by caching remote references. When a processor is working on a row of $\mathbf{M}$ and needs to retrieve a row of $\mathbf{A}$ from another processor it puts that row in a cache (implemented with a hash table). It is very likely that subsequent rows of $\mathbf{M}$ will require the same row of $\mathbf{A}$, which they will find in the cache without resorting to unnecessary communication. The function that accesses rows of $\mathbf{A}$ works as follows:

1. If the row is local simply return it.
2. Otherwise, if it is in the cache return it.
3. Otherwise, initiate a request to the processor that owns it.
4. Service requests until the data arrives and the request queue is empty.
5. Put the row in the cache and return it.

3.3 Load Balancing

It is very likely that some rows of $\mathbf{M}$ will require much more work than the average row, which can lead to a serious load imbalance. Furthermore, it is impossible to predict accurately how much work a row will require, and therefore it is impossible to allocate work to processors ahead of time in a load-balanced distribution. We have implemented a dynamic load balancing strategy to deal with this problem.

Every processor "owns" a number of rows of the matrices $\mathbf{A}$ and $\mathbf{M}$, which are assigned at the outset of the program. The indices of the "local" rows of $\mathbf{M}$ are maintained as a queue and each processor constructs its local part of $\mathbf{M}$ by taking indices from the queue. Suppose processor p reaches the end of the queue, having completed its local work. It sends a message informing the master processor that it has finished its local work, but there

may be other processors which are not finished, so processor p polls the other processors, using the communications server, asking whether they have any row indices of M remaining in their queues. Suppose processor q has such an index. It takes that index from the queue and returns it to processor p, which then computes the row of M in exactly the same way as it would compute a local row of M, and when it is finished it their local work it sends messages (which are handled by the communications server) to the other processors informing them that M is complete.

3.4 User Interface

The SPAI algorithm has a few free parameters that permit the control of the quality of the preconditioner constructed. These parameters specify the number of fill-ins per column, the number of new nonzero entries allowed per step of the algorithm, and ϵ. In spai_1.1, these parameters are called ma, mn and ep, respectively, and we will make use of them in the rest of this paper. spai_1.1 comes bundled with an iterative method (BICGSTAB), and that was the only method we used in this study. Coupling spai_1.1 with other iterative solvers is straightforward.

4 Numerical Experiments

In this Section we present the results we obtained for a set of very diverse sparse matrices, with a number of nonzeros ranging from a few thousand to a couple of million. All the matrices we used can be obtained from the excellent University of Florida Sparse Matrix Webpage maintained by T. Davis [13]. We used matrices from the the HB (Harwell Boeing), Simon, Nasa and Rothberg collections[2]. For practical purposes we will group the matrices according to their sizes. Small matrices will be the ones with less than fifty thousand nonzero entries, medium size matrices will have between fifty thousand and five hundred thousand nonzero entries, and large matrices will have more than five hundred thousand nonzero entries.

Before we present the results, though, we need to settle on some criteria about what is important in those results. There are two issues we have to

[2]In some places, especially when we refer to a matrix for the first time, we will follow the name of the matrix with a "code" like (Nasa,rsa). The first entry refers to the name of the collection to which the matrix belongs, and the second entry to the type of matrix, using HB notation [15]: "rua" refers to unsymmetric matrices and "rsa" to symmetric ones.

consider in order to make a judgement about SPAI. The first is *effectiveness* (by how much can the number of iterations of the iterative method be reduced), and the second is *efficiency* (how long it takes to find the solution, including the time taken evaluating the preconditioner).

All numerical results reported here were obtained on the Cray T3E-600 at NERSC, an MPP system with 176 processors, of which 152 are configured to run parallel computing jobs. The T3E processors are DEC Alphas (EV-5's) with a clock speed of 300 MHz, peak performance of 600 MFlops and 256 MB of memory (but a practical limit of 235 MB for parallel jobs, and 80 MB for jobs in one processor). Hence single processor results listed here are indicative of workstation performance of SPAI. By default, the T3E processors use 64-bit words. Double precision (64 bit) arithmetic was used in all experiments.

4.1 Assessing the Effectiveness of SPAI

A good case in favor of the effectiveness of SPAI was already made in [2] and also in [3, 4]. Here we present more evidence by studying matrices where ILU-type methods either fail or have difficulty. Based on the extensive study of the convergence behavior of ILU preconditioned iterative methods in [14], we selected six matrices where ILU preconditioners either failed or required high levels of fill-in (large k's in ILUT(k)), independently of the iterative solver used, in order to achieve convergence in a small number of steps. These six matrices are listed in Table 1 and the results are displayed in Table 2 and are self explanatory[3]. The *tolerance* was set to 10^{-8} and the iterative method used was BICGSTAB[4]. A right hand side $\mathbf{b}$ of 1's was used, but to better compare with [14] we also tried, for some of the matrices, a right hand side such that the solution is a random vector. No significant difference was observed.

Although SPAI succeeded in some of the matrices where ILU-type methods had failed, the SPAI preconditioner was significantly denser than the ILU-type preconditioners constructed in [14], and it is likely that ILU-type

[3] In the three cases where ILUT(k) preconditioners failed, the following fill-in levels were used [14]: k = 100 for nnc261, k = 44 for nnc1374 and k = 13 for lns3937.

[4] It is well known that sometimes BICGSTAB stagnates. For instance, for nnc261 with the choice of parameters given in Table 2, BICGSTAB reaches the *tolerance* of 10^{-10} after 18 iterations, but then stagnates and never reaches a *tolerance* of 10^{-12}. In all cases presented here, the spai_1.1 parameters or the *tolerance* were chosen in order to avoid that.

methods would have succeeded too if denser preconditioners had been considered. On the other hand, with the exception of lns3937, all the other problems were solved in one processor in a reasonable amount of time. For those cases, the cost of constructing such dense preconditioners seems acceptable (in absolute terms), even on a workstation. And, as we shall see later, the shortest times to solution can in fact be considerably less than the ones shown.

4.2 Assessing the Efficiency of SPAI

As described in Section 3.4, spai_1.1 allows us to choose from a different number of options (parameters), the important ones here being ep, mn and ma (ϵ, maximum number of nonzeros per step of the SPAI algorithm and maximum number of nonzeros per row, respectively). Depending on our choice for those parameters, the final results (sparsity of $\mathbf{M}$, number of iterations of BICGSTAB, but specially the time taken by both the construction of the preconditioner and the iterative method BICGSTAB) can be very different. It is important, therefore, that we have some rule of thumb to decide between the different choice of parameters. This is an issue that needs to be addressed before we decide in favor or against SPAI as an efficient preconditioner (even when run in parallel). Since efficiency is measured by the total time to solution (construction of preconditioner time + iterative method time), the parameters should be chosen so that this time to solution is the shortest possible. This usually happens when the times taken by the preconditioner and iterative method are comparable. To show this fact, we present now some results we obtained with a small set of small matrices. These matrices are by now standard references in the SPAI literature and were also used in [2, 3, 4], and are listed in Table 3. In all cases a right hand side of 1's was used.

For every matrix we will fix a value for ma, usually 5% or 10% of n, the order of the matrix, and we will look at the run times of the preconditioner and the iterative method for different values of ep and mn. The results are displayed in Tables 4–9.

We report now our observations for these six matrices. All the tests were ran in one processor. We recall again that spai_1.1 constructs a left preconditioner. To be consistent with [2, 3] we also used a convergence *tolerance* of 10^{-8}.

orsreg1 The results for this matrix are displayed in Table 4. For larger

Matrix	n	$nnz(\mathbf{A})$	k	BICGSTAB	$\rho_{\mathbf{LU}}$
nnc261	261	1500	–	–	–
nnc666	666	4044	30	> 1000	~ 5
nnc1374	1374	8606	–	–	–
lns131	131	536	1	167	1.3
lns511	511	2796	20	49	6
lns3937	3937	25407	–	–	–

Table 1: Set of matrices that were shown to be very difficult with ILU-type methods [14] and that SPAI solves. The last three columns contain data obtained from [14]: k refers to the ILUT(k) preconditioner used, $\rho_{\mathbf{LU}}$ is the density of the incomplete **LU** matrices relative to **A** and the values in the BICGSTAB column were the number of iterations needed to converge to a *tolerance* of 10^{-8}.

Matrix	ep	ma	SPAI		BICGSTAB	
nnc261	0.4	60	6791	2.17	14	0.09
nnc666	0.4	60	18436	6.62	75	1.18
	0.3	60	22539	8.26	52	0.92
	0.3	101	31914	21.43	45	1.00
nnc1374	0.3	60	48768	19.00	75	2.74
	0.3	101	68302	49.90	67	3.06
	0.2	101	86559	66.01	48	2.65
lns131	0.4	21	1265	0.18	55	0.13
	0.4	51	2050	0.41	36	0.10
	0.4	101	2881	0.89	21	0.07
lns511	0.3	101	21250	12.15	85	1.34
	0.2	101	28125	18.10	77	1.46
	0.2	151	36770	33.95	55	1.25
lns3937	0.1	900	1558045	412.66	1942	154.68
	ep	ma	$nnz(\mathbf{M})$	(sec.)	# iter.	(sec.)

Table 2: The effectiveness of SPAI can be controlled by changing the parameters ep, ma and mn. Here, mn = 5 always, except for lns3937, where a value of mn = 85 was used. The *tolerance* was 10^{-8} in all cases. These results were obtained with one processor, except for lns3937, where 16 processors were used. For these problems SPAI is effective at the cost of constructing a preconditioner with significantly more nonzero entries than the original matrix.

values of ep and mn, no difference is observed between the different cases. This was due to the fact that the value chosen for ma was too large to change the results, as can be seen by the fact that the condition (9) was always satisfied. In fact, a choice for ma of 1% of n, would have given practically the same results. For the cases considered the minimum total time was around 2.15-2.20 seconds.

orsirr2 The results for this matrix are displayed in Table 5. The same comments that applied to orsreg1 apply here. The minimum total time for the cases considered was around 0.90-1.00 seconds.

sherman1 The results for this matrix are displayed in Table 6. The minimum total time for the cases considered was around 0.80-0.90 seconds.

sherman2 The results for this matrix are displayed in Table 7. In this case many more rows did not satisfy condition (9). The minimum total time for the cases considered was around 9-11 seconds, showing in fact that this is a harder problem than the previous ones. Interestingly enough, the minimum times occur for large ep's and large number of iterations.

pores2 The results for this matrix are displayed in Table 8. This was a much harder matrix, as previously noticed [2], and it is suggested there that a left preconditioner makes the problem easier. However, since we also used left preconditioning (spai_1.1 uses left preconditioning), we cannot explain why our results seem to indicate that this matrix is harder than the results of [2, 3] suggest. The minimum total time for the cases considered was around 4 minutes.

saylr4 The results for this matrix are displayed in Table 9. The minimum total time for the cases considered was around 24-25 seconds.

A few but important remarks are worth making now:

1. We didn't find a significant difference between runs with different values of mn and the same ep, for the cases where (9) is almost always satisfied. At most, the results seem to indicate that larger values of mn (but still much smaller than ma) allow a faster evaluation of the preconditioner without real degradation of the convergence rate of BICGSTAB. Also, there are no significant differences between the sparsities of the preconditioners evaluated with different mn's. These

Matrix	n	nnz
orsreg1	2205	14133
orsirr2	886	5970
sherman1	1000	3750
sherman2	1080	23094
pores2	1224	9613
saylr4	3564	22316

Table 3: Set of matrices used to show the dependence of the preconditioner and iterative method times on the parameters ep, mn and ma.

ep	mn	SPAI			BICGSTAB		TTime
0.2	2	33795	12.33	0	23	1.21	13.54
	5	29848	4.76	0	22	0.80	5.56
	10	39755	6.14	0	24	1.07	7.21
0.3	2	11701	1.93	0	37	0.88	2.81
	5	11025	1.53	0	37	0.86	2.39
	10	11025	1.53	0	37	0.89	2.42
0.4	2	11701	1.91	0	37	0.91	2.82
	5	11025	1.54	0	37	0.86	2.40
	10	11025	1.57	0	37	0.95	2.52
0.5	2	8379	1.15	0	49	1.05	2.20
	5	9261	1.19	0	44	0.97	2.16
	10	9261	1.19	0	44	0.96	**2.15**
0.6	2	3969	0.46	0	169	3.04	3.50
	5	3969	0.46	0	169	3.01	3.47
	10	3969	0.46	0	169	3.01	3.47
ep	mn	nnz(M)	(sec.)	$\not\in$	# iter.	(sec.)	(sec.)

Table 4: **orsreg1**: $n = 2205$, $nnz = 14133$. A value of ma $= 55$ was used, which corresponds to 2.5% of n. TTime denotes the total time to solution, and the shortest time is boldfaced. The values in the column labeled by $\not\in$ correspond to the number of rows where (9) was not satisfied.

ep	mn	SPAI			BICGSTAB		TTime
0.2	2	15054	6.35	10	25	0.43	6.78
	5	12699	2.25	0	23	0.37	2.62
	10	17927	2.85	7	20	0.36	3.21
0.3	2	4853	0.93	0	39	0.42	1.35
	5	4417	0.64	0	37	0.37	1.01
	10	4425	0.64	0	38	0.40	1.04
0.4	2	4738	0.91	0	40	0.44	1.35
	5	4329	0.63	0	39	0.39	1.02
	10	4329	0.63	0	39	0.39	1.02
0.5	2	3266	0.48	0	56	0.52	1.00
	5	3626	0.50	0	46	0.45	0.95
	10	3626	0.49	0	46	0.43	**0.92**
0.6	2	1566	0.20	0	231	1.88	2.08
	5	1566	0.20	0	231	1.87	2.07
	10	1566	0.20	0	231	1.86	2.06
ep	mn	$nnz(\mathbf{M})$	(sec.)	$\not\ell$	# iter.	(sec.)	(sec.)

Table 5: **orsirr2**: $n = 886$, $nnz = 5970$. A value of ma $= 44$ was used, which corresponds to 5% of n.

observations seem to disagree with [3], but are also inconclusive since the number of experiments was quite small. We decided not to pursue this further.

2. In the cases where a larger number of rows violate (9), there is stronger evidence that larger values of mn allow a faster evaluation of the preconditioner at the cost of increasing the number of iterations needed by BICGSTAB to converge. The total times are not necessarily larger though.

3. The minimum total time usually occurs when the time taken to evaluate the preconditioner is very close to the time taken by the iterative method (BICGSTAB) to converge to the required tolerance. This also means that the corresponding number of iterations can be large.

4. The parameter ep is the most important one, as expected.

Of the four above remarks, the third one is the most important one and the key to speed up the search for the shortest time to solution; *i.e.*, looking

ep	mn	SPAI			BICGSTAB		TTime
0.2	2	15520	5.83	47	22	0.34	6.17
	5	14715	2.21	23	23	0.35	2.56
	10	15757	1.87	24	20	0.31	2.18
0.3	2	6690	1.45	6	34	0.36	1.81
	5	7329	0.82	8	34	0.39	1.21
	10	8136	0.77	8	31	0.36	1.13
0.4	2	4437	0.92	6	45	0.40	1.32
	5	4874	0.54	5	45	0.42	0.96
	10	5051	0.49	4	40	0.39	0.88
0.5	2	2721	0.38	0	79	0.61	0.99
	5	3333	0.36	2	59	0.49	0.85
	10	3407	0.34	2	57	0.47	**0.81**
0.6	2	1791	0.25	0	106	0.76	1.01
	5	2021	0.24	2	102	0.73	0.97
	10	2031	0.23	2	84	0.65	0.88
ep	mn	$nnz(\mathbf{M})$	(sec.)	∉	# iter.	(sec.)	(sec.)

Table 6: **sherman1**: $n = 1000$, $nnz = 3750$. A value of ma $= 50$ was used, which corresponds to 5% of n.

ep	mn	SPAI			BICGSTAB		TTime
0.4	5	14696	18.96	42	32	0.84	19.80
	10	14700	11.44	24	53	1.53	12.97
	20	14007	7.97	75	335	9.12	17.09
0.5	5	13518	16.17	24	34	0.87	17.04
	10	13581	9.07	18	65	1.68	10.75
	20	13074	6.76	63	338	8.64	15.40
0.6	5	12646	14.51	20	39	0.98	15.49
	10	12547	8.10	16	72	1.79	9.89
	20	12324	6.14	52	195	4.84	10.98
0.7	5	10830	10.90	8	53	1.28	12.18
	10	10940	6.26	10	108	2.62	**8.88**
	20	10767	5.01	39	555	13.42	18.43
ep	mn	$nnz(\mathbf{M})$	(sec.)	∉	# iter.	(sec.)	(sec.)

Table 7: **sherman2**: $n = 1080$, $nnz = 23094$. A value of ma $= 54$ was used, which corresponds to 5% of n.

ep	mn	SPAI			BICGSTAB		TTime
0.2	5	198865	391.48	999	898	97.30	488.78
	10	197314	220.70	1021	1339	147.15	367.85
	20	190636	150.23	1041	1526	159.42	309.65
0.3	5	142305	232.41	441	1269	103.80	336.21
	10	147205	135.20	426	1077	91.61	226.81
	20	154088	104.84	503	2326	202.52	307.36
0.4	5	82530	100.15	107	4366	232.51	332.66
	10	90239	65.95	113	3096	182.85	248.80
	20	102022	55.19	145	2219	138.46	**193.65**
ep	mn	nnz(M)	(sec.)	$\not\ell$	# iter.	(sec.)	(sec.)

Table 8: **pores2**: $n = 1224$, $nnz = 9613$. A value of ma $= 183$ was used, which corresponds to 15% of n.

ep	mn	SPAI			BICGSTAB		TTime
0.2	5	84074	48.66	13	72	5.29	53.95
	10	87883	30.18	31	72	5.38	35.56
	20	84873	22.72	92	77	5.65	28.37
0.3	5	45566	12.74	0	222	12.46	25.20
	10	47202	9.73	3	310	17.19	26.92
	20	47044	8.53	4	271	15.00	**23.53**
0.4	5	26310	6.83	0	741	30.88	37.71
	10	26459	4.59	3	583	24.26	28.85
	20	25816	3.73	4	609	25.15	28.88
ep	mn	nnz(M)	(sec.)	$\not\ell$	# iter.	(sec.)	(sec.)

Table 9: **saylr4**: $n = 3564$, $nnz = 22316$. A value of ma $= 178$ was used, which corresponds to 5% of n.

for the point in parameter space where the times taken by the preconditioner and iterative method are comparable is a good way to look for the shortest time to solution.

This may mean in general that the corresponding number of iterations can be quite large. From the perspective of a Numerical Linear Algebra theorist this may seem a displeasing choice, but from the perspective of a user that needs a preconditioner to solve problems quickly, this is the right choice because it usually leads to the shortest times to solution. It is not difficult to give a heuristic explanation why that is so.

If we plot the times taken to construct a preconditioner versus some measure of the "quality" of that preconditioner (like ep, where smaller values for ep mean better quality), we can intuitively expect that the resulting plot corresponds to a decreasing and convex function of ep. Similarly, intuitively, we can expect the plot of the times taken by the iterative method to converge to a required tolerance, to be an increasing and convex function of ep. If we consider now the total times to solution (*i.e.*, the sum of the two plots), then intuitively we expect the minimum to be close to the point where the two plots meet, *i.e.*, the point where the time to construct the preconditioner and the time for the iterative method to converge are "roughly" the same. As an example, we consider the matrix sherman1 with mn = 5 and ep = 0.2:0.1:0.6 (Matlab notation). A graph of bars is shown in Fig. 1.

In [3], the SPAI and ILU preconditioners were compared by looking at the times to construct the preconditioners that give roughly the same number of iterations. Although that effectively compares the costs associated with the two preconditioners (and the verdict was that SPAI is very expensive relatively to ILU, in one processor[5]), it doesn't guarantee that we are looking at the shortest times to solution. As a matter of fact though, the ILU times given in [3] are comparable to their respective iterative methods times, and probably the corresponding solution times are very close to the shortest times to solution.

There are situations (*e.g.*, when various right hand sides are present) where chosing the parameters so that the times spent in constructing the preconditioner and in the iterative method are roughly the same is not a good choice (but extending that argument further, if we have even more right hand sides, direct methods end up being cheaper than iterative methods). And finally, there is no way to determine the point in parameter space with the shortest time to solution except by trial and error (a time-consuming

[5]The same conclusion was reached in [4].

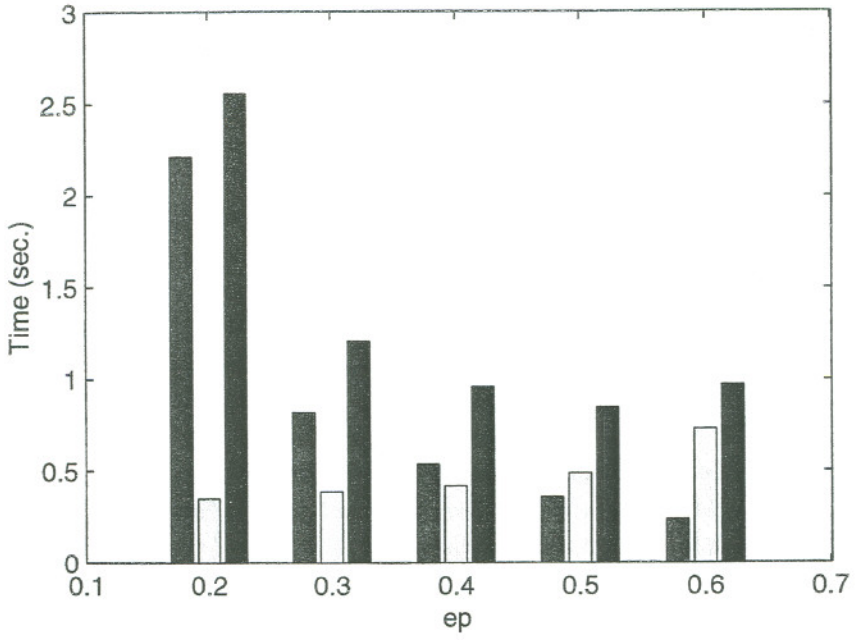

Figure 1: **sherman1**: Time taken to construct preconditioner with SPAI (left bar); time taken by BICGSTAB to converge to *tolerance* of 10^{-8} (middle bar); total time (right bar), as a function of ep. The times are given in seconds, and were obtained with ma = 44 and mn = 5, as given in Table 6.

task), and there is no guarantee that there is only one such point, except for a heuristic argument. Keeping this in mind, we decided nevertheless to present mostly the results corresponding to the shortest times to solution (where the time spent in constructing the preconditioner and the time spent in the iterative method are roughly the same), even if the number of iterations may seem very large.

4.3 The Experiments

The results presented below correspond to the shortest times to solution (among the few tests we performed). As stressed in the previous section, we could present results corresponding to more effective preconditioners (smaller number of iterations), but those preconditioners would be less efficient (longer times to solution).

In general, the results for small matrices were obtained with only one processor, while for medium size and large matrices we used multiple processors. Here we report only cases that SPAI solved, and we will leave the few cases where it failed to Section 6. Again, we used in all cases a right hand side of 1's. In the few cases where a right hand side was provided, we also repeated (some of) the experiments with the given right hand side, but no significant differences were observed.

4.3.1 Small Matrices

The results for matrices (HB,rua) with less than fifty thousand nonzero entries are presented in Table 10, and are self explanatory. To be consistent with [2, 3] (although we always use a left preconditioner) we set the *tolerance* to 10^{-8} (in [4] a value of 10^{-9} was used). For sherman3 the default starting vector $\mathbf{x}_0$ and the right hand side $\mathbf{b}$ were such that the *tolerance* was satisfied after the first iteration. This was accidental and not representative of the method.

4.3.2 Medium Size Matrices

These are the matrices with more than fifty thousand but less than five hundred thousand nonzero entries. The results are presented in Table 11 for some bcsstk (HB,rsa) and raefsky (Simon,rua) matrices, and are self explanatory[6]. The *tolerance* was set to 10^{-10}. Figure 2 refers to raefsky2

[6]Since the matrix raefsky1 was also studied in [3], it is worth comparing the two results, even though we used a left preconditioner while in [3] a right preconditioner was used. The fact that raefsky1 seems to be only slightly unsymmetric, at least visually (see Section 6), makes such comparison meaningful.

The following table is self explanatory. `MI12` is the code used in [3].

Code	ma	ep	mn	$nnz(\mathrm{M})/nnz(\mathbf{A})$	SPAI	BICGSTAB	
MI12	50	0.3	1	0.087	665.45	86	22.64
spai_1.1	50	0.3	1	0.0867	475.90	87	14.19
spai_1.1	21	0.5	5	0.033	16.20	127	20.30
	ma	ep	mn	$nnz(\mathrm{M})/nnz(\mathbf{A})$	(sec.)	# iter.	(sec.)

A *tolerance* of 10^{-8} was used. The last row, which corresponds to the shortest time to solution, clearly shows how the efficiency can be greatly improved by choosing the right parameters. For the second row, we use the parameters used in [3, 2]. The ratios MI12/spai_1.1 of the SPAI CPU times and BICGSTAB CPU times are 1.40 and 1.51, respectively (and are thus consistent). Evidently, the spai_1.1 results were obtained in one processor.

Matrix	n	$nnz(\mathbf{A})$	SPAI		BICGSTAB	
orsreg1	2205	14133	9261	1.19	44	0.96
orsirr1	1030	6858	4326	0.54	44	0.47
orsirr2	886	5970	3626	0.49	46	0.43
sherman1	1000	3750	3407	0.34	57	0.47
sherman2	1080	23094	10940	6.26	108	2.62
sherman3	5005	20033	5005	0.41	1	0.04
sherman4	1104	3786	1104	0.10	64	0.47
sherman5	3312	20793	11155	1.45	53	1.71
saylr3*	1000	3747	4510	0.40	57	0.54
saylr4	3564	22316	47044	8.53	271	15.00
pores2	1224	9613	102022	55.19	2219	138.46
lnsp3937	3937	25407	275107	40.58	3262	96.54
watt2	1856	11550	155686	29.82	1742	28.88
	n	$nnz(\mathbf{A})$	$nnz(\mathbf{M})$	(sec.)	# iter.	(sec.)

Table 10: The results correspond to the shortest times to solution among the limited number of tests we performed. Here we used *tolerance* $= 10^{-8}$ and 1 processor, except for lnsp3937 and watt2 where 8 processors were used. saylr3* was obtained from saylr3 by replacing two independent 2×2 singular submatrices by 2×2 identity matrices, as explained in [2], pag. 18.

(Simon,rua), another medium size matrix. It shows variations in the number of iterations as ep changes. This is a typical behavior in that smaller values of ep mean better preconditioners and fewer iterations. However, in this case, the shortest time happens for ep = 0.80 (incidentally, raefsky2 is diagonal dominant and very easy to solve). Table 13, in Section 5, shows the scaling properties of spai_1.1 for the matrix bcsstk17.

4.3.3 Large Matrices

These are the matrices with more than five hundred thousand nonzero entries. The results, obtained for a *tolerance* of 10^{-10} are shown in Table 12 for raefsky3 (Simon,rua) and the cfd matrices (Rothberg,rsa), and are again self explanatory. As we can see from the results, SPAI can be an expensive method, even when restricted to large problems in large number of processors. As expensive as it can seem (at least for cfd2), it is probably still more efficient than any other method, but we do not have data to support this

Matrix	n	$nnz(\mathbf{A})$	SPAI		BICGSTAB	
bcsstk14	1806	63454	30733	3.74	120	1.29
bcsstk16	4884	290378	11094	1.54	39	1.05
bcsstk17	10974	428650	261849	593.41	1227	87.09
raefsky1	3242	294276	9812	3.68	145	3.75
raefsky5	6316	168658	23221	1.16	10	1.27
raefsky6	3402	137845	16594	3.75	151	2.49
	n	$nnz(\mathbf{A})$	$nnz(\mathbf{M})$	(sec.)	# iter.	(sec.)

Table 11: The results correspond to the shortest times to solution among the limited number of tests we performed. Here we used *tolerance* $= 10^{-10}$ and 8 processors, except for raefsky5 where only one processor was used. For bcsstk17 we were unable to find a better SPAI-BICGSTAB time ratio (BICGSTAB would always break down for any choice of parameters likely to push the ratio in the right direction.).

Matrix	n	$nnz(\mathbf{A})$	SPAI		BICGSTAB	
raefsky3	21200	1488768	859892	307.9	3328	342.3
cfd1	70656	1828364	1327276	112.4	889	171.0
cfd2	123440	3087898	5272790	418.6	2714	668.7
	n	$nnz(\mathbf{A})$	$nnz(\mathbf{M})$	(sec.)	# iter.	(sec.)

Table 12: The results correspond to the shortest times to solution among the limited number of tests we performed. Here we used *tolerance* $= 10^{-10}$ and 16 processors, except for cfd2 where 64 processors were used. In Figure 5 the scaling behavior of spai_1.1 for cfd2 is shown.

claim (we did not find any published results to which we could compare).

5 Parallel Performance of spai_1.1

In this Section we discuss the scaling properties of spai_1.1, both during the construction of the preconditioner and during the iterative phase. Two examples will be considered: bcsstk17 and cfd2.

The results for bcsstk17 are displayed in Table 13 and Figures 3 and 4. This matrix required a minimum of two processors. It is clear from the Figures that SPAI scales considerably better (at least on the T3E) than does BICGSTAB. This scaling will be sensitive, however, to the latency of

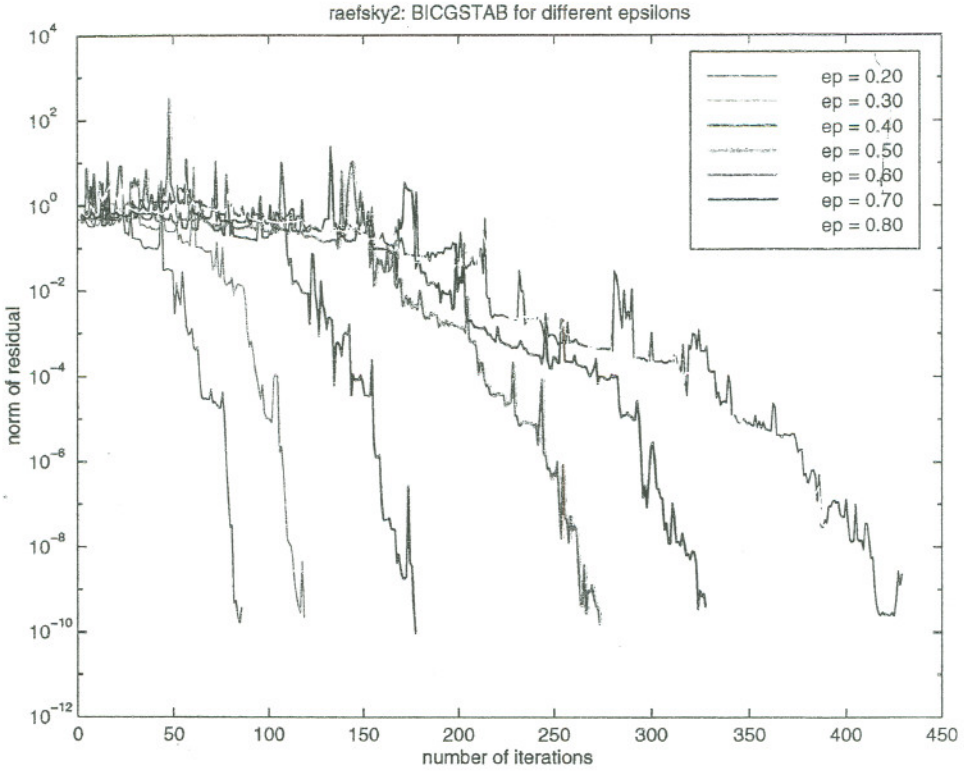

Figure 2: **raefsky2**: $n = 3242$, $nnz = 294276$. Effectiveness of the preconditioner increases (*i.e*, number of iterations decreases) as ep decreases. In this case the shortest total time happens for ep = 0.80, and the preconditioner is just diagonal.

interprocessor communication (which is very good on the T3E). The scaling of BICGSTAB (or any iterative solver) is limited by remote references incurred in the inner products. The sparse matrix-vector multiply routine used by the BICGSTAB solver in spai_1.1 attempts to hide this latency by overlapping local work with communication.

The results for cfd2 are displayed in Table 14 and Figure 5. This matrix did not fit in 8 processors with the choice of parameters used.

# procs	2	4	8	16	32	64
SPAI (sec.)	2120.1	1114.1	593.4	327.8	184.0	103.2
BICGSTAB (sec.)	275.9	155.7	87.1	53.7	38.4	32.4
BICGSTAB (# iter.)	1252	1191	1227	1255	1270	1135

Table 13: **bcsstk17**: Scaling of spai_1.1, including the iterative phase. A *tolerance* of 10^{-10} was used, and for the set of parameters used, spai_1.1 constructs a preconditioner with $nnz(\mathbf{M}) = 261849$. The number of iterations depends on the number of processors, a well known fact that hinders the study of the scalability of BICGSTAB (or any other iterative method).

# procs	16	32	64	128
SPAI (sec.)	1228.2	707.4	418.4	233.2
BICGSTAB (sec.)	1068.2	830.0	667.0	453.5
BICGSTAB (# iter.)	2379	2740	2714	2319

Table 14: **cfd2**: Scaling of spai_1.1, including the iterative phase. A *tolerance* of 10^{-10} was used, and for the set of parameters used, spai_1.1 constructs a preconditioner with $nnz(\mathbf{M}) = 5272790$.

6 Topics on SPAI

In this Section we discuss miscellaneous topics concerning the quality of the preconditioner $\mathbf{M}$ constructed by SPAI, and why sometimes it totally fails as a preconditioner. Some of these issues were already discussed in [2, 3].

It was shown in [2] that SPAI is very effective át capturing the sparsity pattern of the real inverse. This was concluded after comparing the portraits of $\mathbf{M}$ and $\tilde{\mathbf{A}}$ (we define this matrix as being the one obtained from $\mathbf{A^{-1}}$ by keeping its largest entries, in absolute value, and in the same number as the number of nonzero entries in $\mathbf{M}$). However, this picture can be misleading since there is no guarantee that $\tilde{\mathbf{A}}$ is a good preconditioner. As a matter of fact, usually it's not. Also, sometimes SPAI fails to get the sparsity structure of $\tilde{\mathbf{A}}$, but $\mathbf{M}$ is nevertheless a good preconditioner. This point is illustrated in Figure 6.

To study how good $\tilde{\mathbf{A}}$ could be as a preconditioner, we submitted $\tilde{\mathbf{A}}$ to a couple of tests where $\mathbf{M}$ does well by construction. We performed those tests with a couple of matrices and the results were sufficiently consistent to show that in general $\tilde{\mathbf{A}}$ is a bad preconditioner. Here we report the results obtained with the by now popular orsirr2 matrix. Figure 7 shows that in

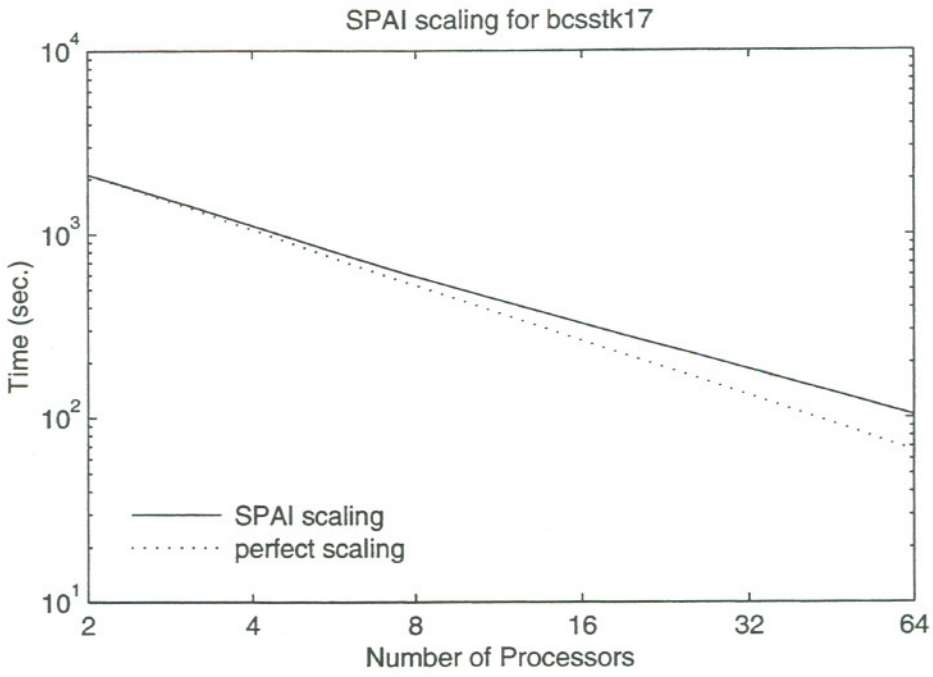

Figure 3: **bcsstk17**: SPAI scales linearly with the number of processors. For 64 processors, there is a performance degradation of 55% relative to perfect scaling (from 2 processors).

this case the preconditioner $\mathbf{M}$ captures the sparsity of $\tilde{\mathbf{A}}$ rather well and is also a good preconditioner. Notice how by an appropriate choice of spai_1.1 parameters we obtained a much sparser preconditioner than in [2]. Since by construction, $\mathbf{M}$ minimizes the Frobenius norm of $(\mathbf{MA} - \mathbf{I})$, we decided to see how well $\tilde{\mathbf{A}}$ would compare. The results are shown at the top of Figure 8. If minimizing Frobenius norm is a necessary condition to have a good preconditioner, then it's clear that $\tilde{\mathbf{A}}$ will be a bad preconditioner. To confirm that, we evaluated the eigenvalues of both $\mathbf{MA}$ and $\tilde{\mathbf{A}}\mathbf{A}$. The results are displayed in Figure 9. As a last test we used $\tilde{\mathbf{A}}$ as a preconditioner with BICGSTAB and the bad qualities of $\tilde{\mathbf{A}}$ as a preconditioner were again confirmed ($\tilde{\mathbf{A}}$ turned out to be a matrix "close to singular or badly scaled" and the iterative method broke down). At the bottom of Figure 8 we also compare the number of nozero entries per row for $\mathbf{M}$ and $\tilde{\mathbf{A}}$.

Among all the matrices we tried there were a few where SPAI failed, even with all the leverage that a T3E provides. An example of a small matrix

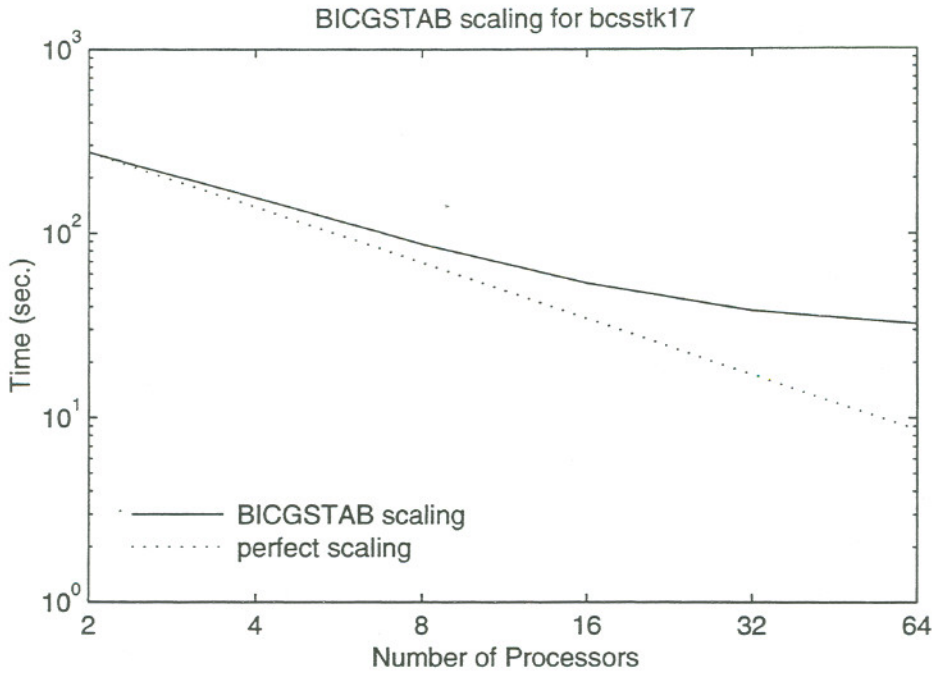

Figure 4: **bcsstk17**: BICGSTAB "scales" nonlinearly with the number of processors. The high costs of communication are evident in the convexity of the scaling "function".

where that happens, is gre1107 (HB,rua), Figure 10. We think that is mainly due to the high degree of assymmetry in $\mathbf{A}$. That this was a difficult matrix had already been noticed in [3]. In this case, as in all other cases where we failed to solve $\mathbf{Ax} = \mathbf{b}$, we were limited by memory constraints. As examples of large matrices where SPAI failed, we have raefsky4 (Simon,rua) and nasasrb (Nasa,rsa). The matrix raefsky4 is particularly difficult because the largest entries (in absolute size) occur far from the diagonal, as can be seen[7] in Figure 11. The same happens with lns3937, Figure 12, but due to its smaller size it was possible to find a solution by chosing a large values for ma (this seems to be a necessity when the largest entries are far from the diagonal), as we saw in Table 2. On the other hand, nasasrb does not seem to be very difficult just by visual inspection. However, its size greatly reduced our capabilities to tweak with the parameters within the memory

[7]The color plates and the statistics information were obtained from the University of Florida Sparse Matrix Webpage [13].

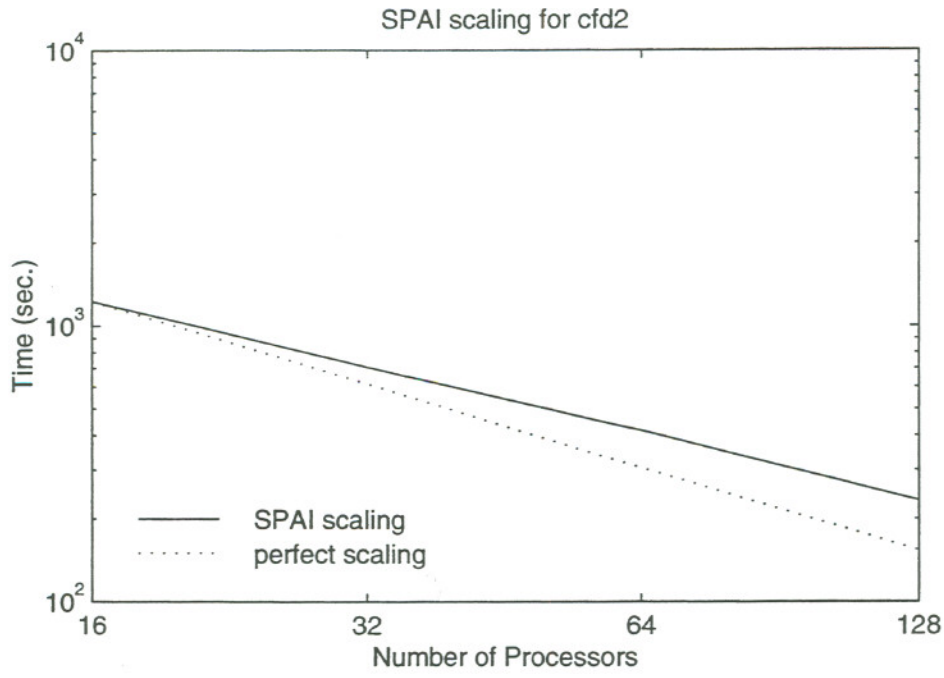

Figure 5: **cfd2**: SPAI scales linearly with the number of processors. For the choice of parameters used, 8 processors were not enough to hold this matrix. For 128 processors, there is a performance degradation of 52% relative to perfect scaling (from 16 processors).

constraints, and we were unable to see any sign of convergence.

Visual inspection of the absolute size of the matrix entries, either by using the Emily Visualization Tool, or just by looking at the University of Florida Sparse Matrix Webpage (as we did), turned out to be a very useful way to quickly guess what parameters to use in spai_1.1 and how well SPAI could perform. As an example, it was easy to predict that SPAI would solve cfd1 (Rothberg,rsa) quite easily (considering its size) due to the "visual" dominance of the diagonal (Figure 13), and the tests confirmed that. Just a knowledge of the pattern of the entries would not allow such conclusions (compare, say, cfd1 with bcsstk14 (HB,rsa), whose pattern is similar, but turned out to be a much harder matrix, considering its smaller size). Similarly we guessed SPAI would do rather well with cfd2, the largest matrix we tried, and the tests confirmed that too.

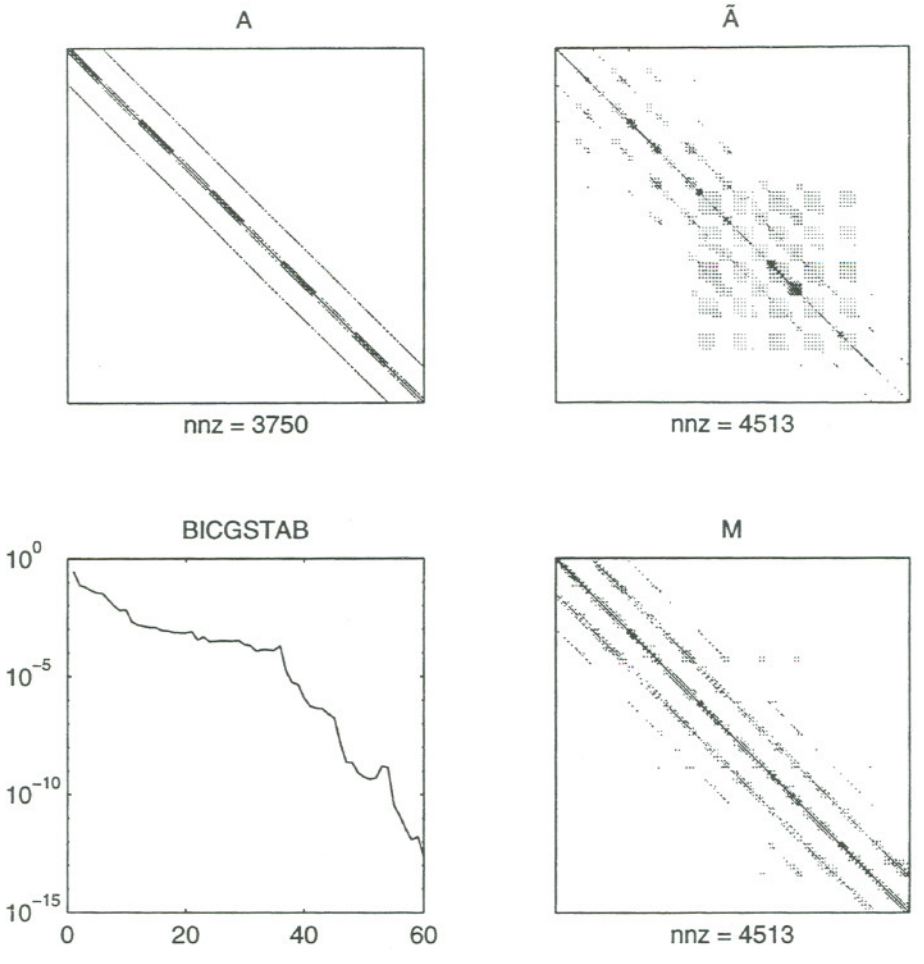

Figure 6: **sherman1**: Although SPAI fails to capture important features of $\tilde{\mathbf{A}}$, $\mathbf{M}$ is nevertheless a good preconditioner. Upper left: $\mathbf{A}$; upper right: $\tilde{\mathbf{A}}$; lower right: $\mathbf{M}$; lower left: BICGSTAB plot. For $\mathtt{ep} = 0.4, \mathtt{ma} = 21, \mathtt{mn} = 5$ and $tol. = 10^{-12}$, SPAI computes $\mathbf{M}$ in 0.41 secs and BICGSTAB converges in 60 iterations and 0.45 secs.

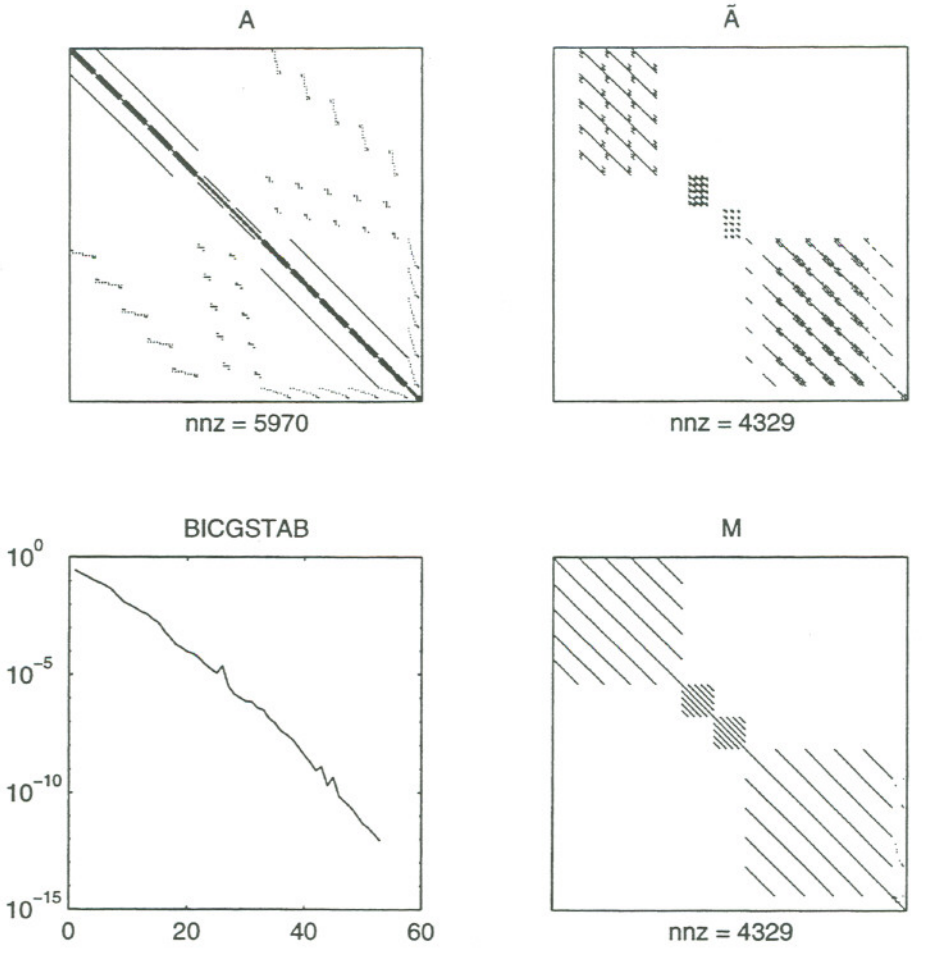

Figure 7: **orsirr2**: In this case SPAI captures many of the features of $\tilde{\mathbf{A}}$. M is also a good preconditioner. Upper left: A; upper right: $\tilde{\mathbf{A}}$; lower right: M; lower left: BICGSTAB plot. For ep = 0.4, ma = 21, mn = 5 and *tol.* = 10^{-12}, SPAI computes M in 0.62 secs and BICGSTAB converges in 53 iterations and 0.40 secs.

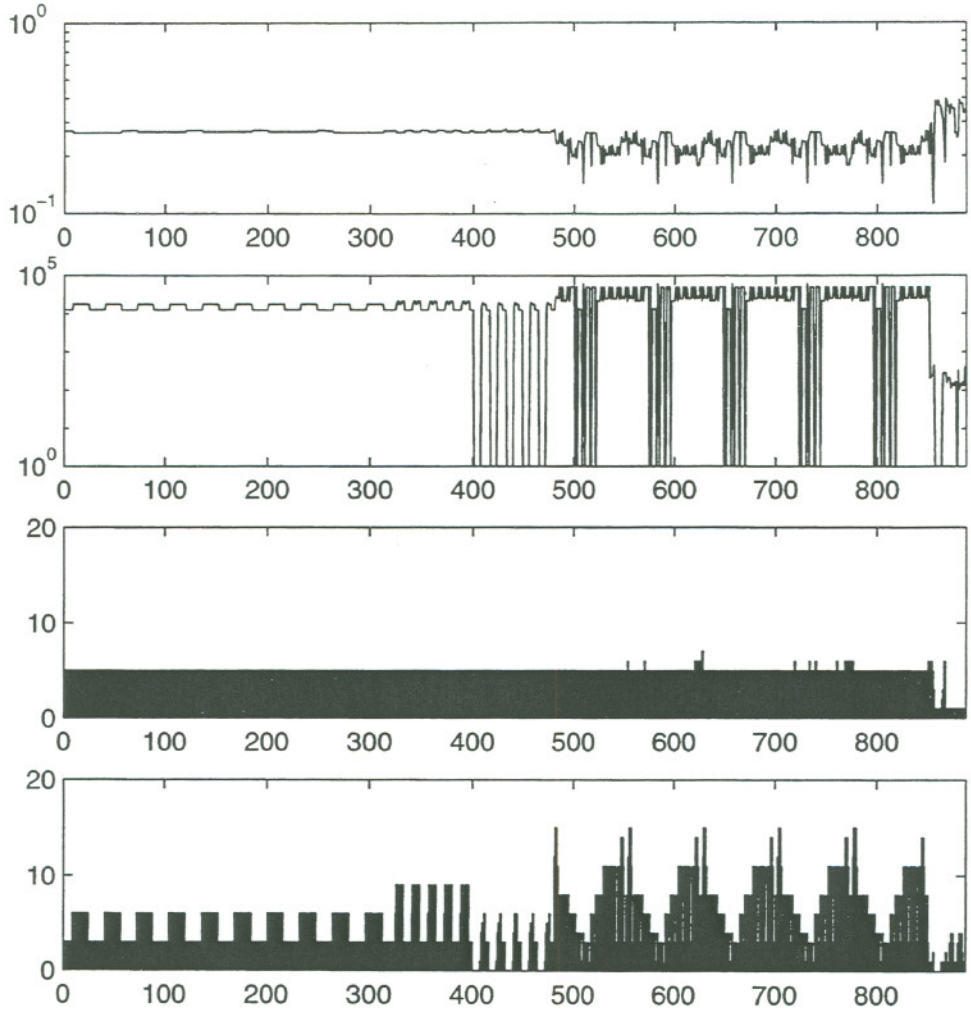

Figure 8: **orsirr2**: Here we compare $\mathbf{M}$ with $\tilde{\mathbf{A}}$. The top two plots compare the Frobenius norm of the rows of $(\mathbf{MA} - \mathbf{I})$ (1st plot), and $(\tilde{\mathbf{A}}\mathbf{A} - \mathbf{I})$ (2nd plot). The two bottom plots compare the number of nonzeros per row of $\mathbf{M}$ (3rd plot) and $\tilde{\mathbf{A}}$ (4th plot). The spai_1.1 parameters (ma, mn and ep) are the same as in Figure 7. For those values we obtained $\|\mathbf{MA} - \mathbf{I}\|_F = 7.5736$ and $\|\tilde{\mathbf{A}}\mathbf{A} - \mathbf{I}\|_F = 7.8734 \times 10^5$.

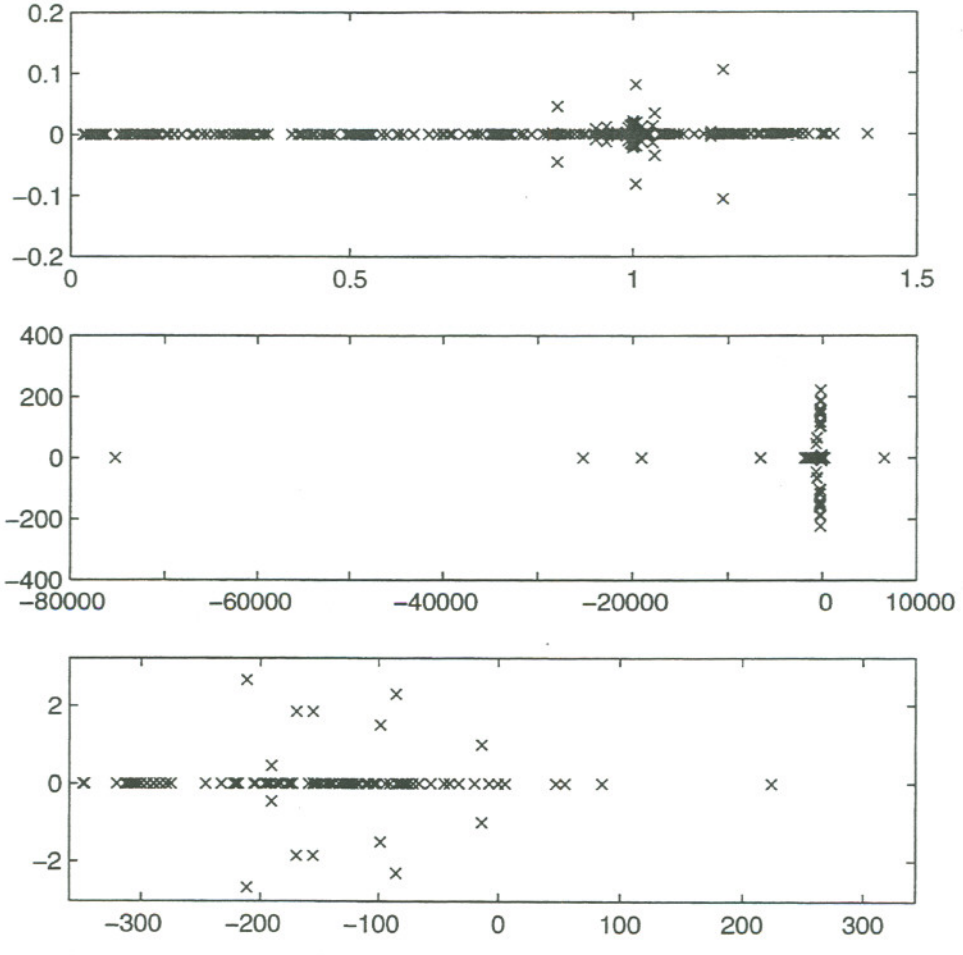

Figure 9: **orsirr2**: Here we compare the eigenvalues of $\mathbf{MA}$ (top plot) and $\tilde{\mathbf{A}}\mathbf{A}$ (middle plot). The bottom plot is a zoom in of the middle one. This gives clear evidence that $\tilde{\mathbf{A}}$ would be a very poor preconditioner.

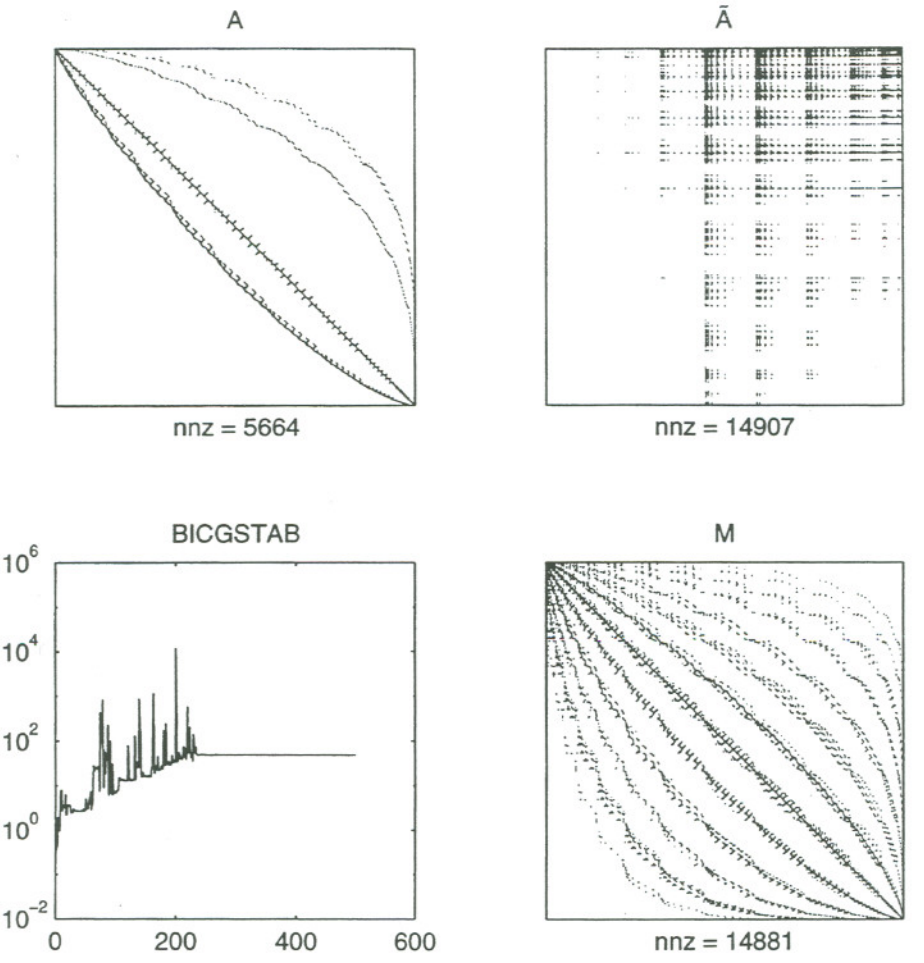

Figure 10: **gre1107**: $n = 1107$, $nnz = 5664$. In this case $\mathbf{M}$ and $\tilde{\mathbf{A}}$ have nothing in common. This matrix, although rather small, resisted all attempts, and was never solved. The above plots were obtained with ep = 0.4, ma = 21 and mn = 5.

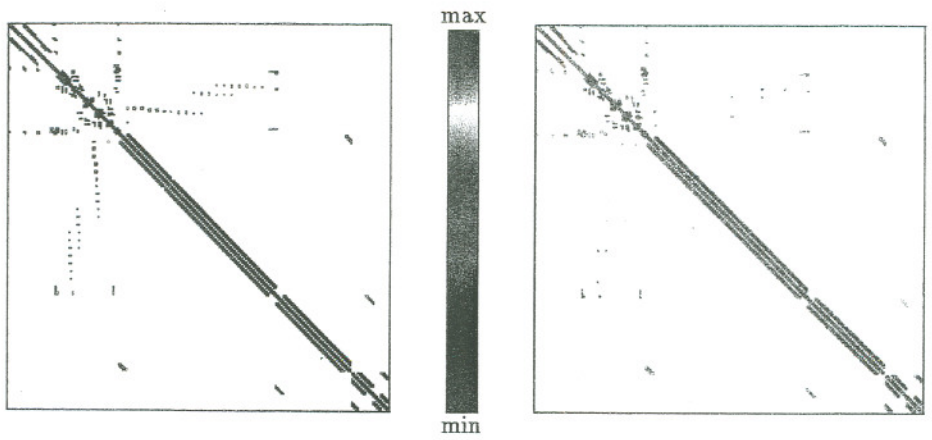

Figure 11: **raefsky4**: This was a matrix where SPAI failed, because the largest entries occur far from the diagonal. Statistics: $n = 19779$, $nnz(\mathbf{A}) = 1328611$, max $= 1.568 \times 10^{11}$, min $= 8.882 \times 10^{-16}$.

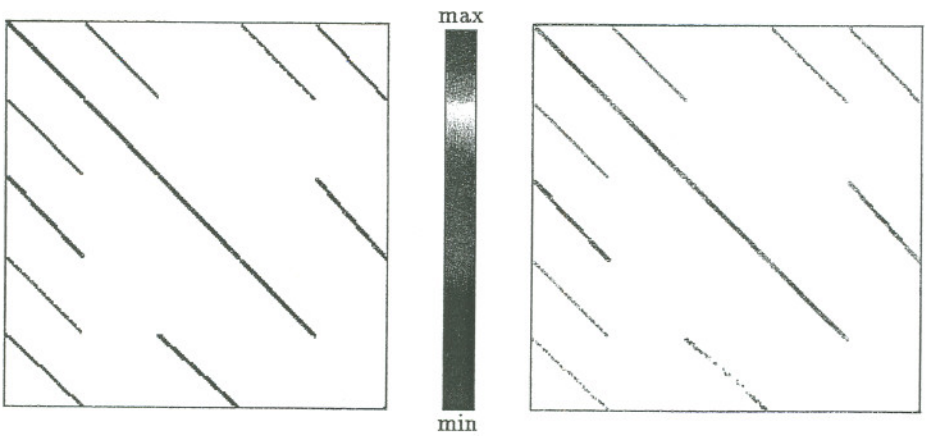

Figure 12: **lns3937**: Although the largest entries occur far from the diagonal, the matrix was sufficiently small to be solved within the memory constraints of the T3E. Statistics: $n = 3937$, $nnz(\mathbf{A}) = 25407$, max $= 1.938 \times 10^{11}$, min $= 3.203 \times 10^{-7}$.

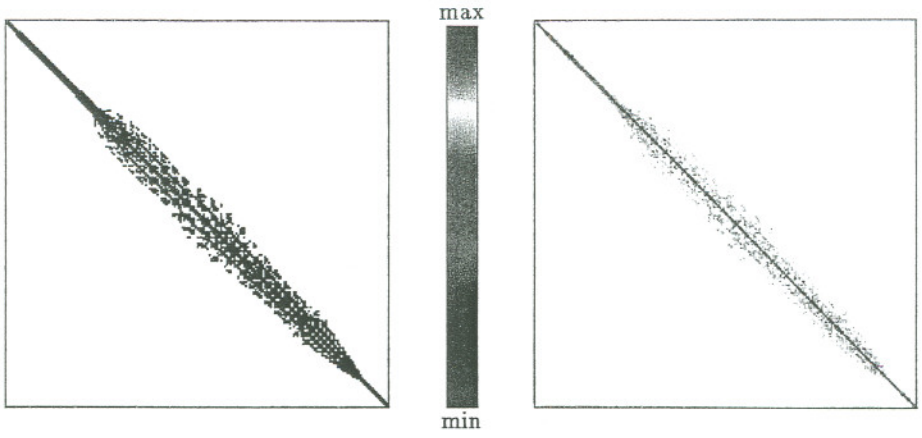

Figure 13: **cfd1**: Although quite large, the dominance of the diagonal suggested this would be a relatively easy matrix, and the tests confirmed that. Statistics: $n = 70656$, $nnz(\mathbf{A}) = 1828364$, max $= 1$, min $= 7.824 \times 10^{-7}$.

7 Conclusions

The SPAI preconditioner is very effective, and almost any problem can be solved by an appropriate choice of parameters (in spai_1.1 those parameters are ma, mn and ep). However, it can be very expensive, even in the very large problems that were supposed to be the most appropriate for a parallel preconditioner like SPAI. Evidently, the more effective the preconditioner, the more expensive it is, but in general the costs are such that shortest times to solution (*i.e.*, time taken to construct preconditioner + time taken by iterative method) require a poor preconditioner. Nevertheless, SPAI (or spai_1.1) is probably the best choice available when trying to solve very large problems.

In general, the choice of an appropriate preconditioner (either effective or efficient) requires extensive tuning between the free parameters (ma, mn and ep). On the other hand, the preconditioners corresponding to the shortest times to solution have, in general, fewer nonzero entries than the original matrix. For difficult matrices, the situation is the opposite, with preconditioners having significantly more nonzero entries than the original matrix.

spai_1.1 is an implementation of SPAI with good load balance and scaling properties. This also shows that the expensiveness of the method is inherent to SPAI and not to the implementation.

References

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Publications, 1994

[2] M.J. Grote and T. Huckle, *Parallel Preconditioning with Sparse Approximate Inverses*, SIAM J. Sci. Comp., 18:838-53, 1997.

[3] N.I.M. Gould and J.A. Scott, *On Approximate-inverse Preconditioners*, Technical Report RAL-95-026, Computing and Information Systems Department, Atlas Centre, Rutherford Appleton Laboratory, Oxfordshire, England, June 1995.

[4] M. Benzi and M. Tuma, *A Sparse Approximate Inverse Preconditioner for Nonsymmetric Linear Systems*, SIAM J. Sci. Comp., in press.

[5] S.T. Barnard and R.L. Clay, *A Portable MPI Implementation of the SPAI Preconditioner in ISIS++*, in Proc. Eight SIAM Conference for Parallel Processing for Scientific Computing, March 1997.

[6] L.Yu. Kolotilina, A.A. Nikishin and A.Yu. Yeremin, *Factorized Sparse Approximate Inverse (FSAI) Preconditionings for Solving 3D FE Systems on Massively Parallel Computers II*, in R. Beauwens and P. de Groen, editors, Iterative Meth. in Lin. Alg., Proc. of the IMACS Internat. Sympos., Brussels, pages 311-312, 1991.

[7] Ju.B. Lifshitz, A.A. Nikishin and A.Yu. Yeremin, *Sparse Approximate Inverse Preconditionings for Solving 3D CFD Problems on Massively Parallel Computers*, in R. Beauwens and P. de Groen, editors, Iterative Meth. in Lin. Alg., Proc. of the IMACS Internat. Sympos., Brussels, pages 83-84, 1991.

[8] L.Yu. Kolotilina and A.Yu. Yeremin, *Factorized Sparse Approximate Inverse Preconditionings*, SIAM Journal on Matrix Analysis and Applications, 14(1):45-58, 1993.

[9] M. Grote and H. Simon, *Parallel Preconditioning and Approximate Inverses on the Connection Machine*, in Proc. of the Scalable High Performance Computing Conference (SHPCC), Williamsburg, VA, pages 76-83, IEEE Comp. Sci. Press, 1992.